Feb 10, 2026 • 11 min read

Laravel AI SDK Tutorial: Build a Smart Assistant in 30 Minutes

Build a working document analyzer with structured output, streaming, and tests using Laravel's new AI SDK. Step-by-step tutorial.

Laravel AI SDK PHP Tutorial OpenAI

Taylor Otwell and Josh Cirre just dropped a 2-hour livestream building AI apps with the Laravel AI SDK. I watched the whole thing. The short version? This SDK is more practical than most tutorials make it look.

I wrote a breakdown of what the AI SDK changes and why it matters when it launched last week. That post covers the "should I use this?" question. This one is the hands-on part.

response back in real time. You'll have a working AI feature in your Laravel app in about 30 minutes.

Let's get into it.

What You'll Need

Nothing crazy here:

- → Laravel 12.x (fresh install or existing project)
- → PHP 8.2+
- → An OpenAI API key (or Anthropic, Gemini, whatever you prefer)

I'm using OpenAI in the examples, but that's the whole point of this SDK. You can swap providers with a single line change. More on that later.

Installation

```
composer require laravel/ai
php artisan vendor:publish --provider="Laravel\Ai\AiServiceProvider"
php artisan migrate
```

Copy

That last command is important. The SDK creates `agent_conversations` and `agent_conversation_messages` tables for conversation memory. Skip the migration and you'll get a confusing table-not-found error the first time you try anything conversational.

Now add your API key to `.env`:

```
OPENAI_API_KEY=sk-your-key-here

# Or if you prefer Anthropic:
# ANTHROPIC_API_KEY=your-key-here
```

Copy

Quick tip: if you don't want to pay for API calls while experimenting, install Ollama and pull a local model. The SDK supports it as a provider. Free and fast for testing.

This is where it gets fun. One artisan command:

```
php artisan make:agent DocumentAnalyzer
```

Copy

This creates `app/Ai/Agents/DocumentAnalyzer.php`. Here's what the generated class looks like with our instructions filled in:

```
<?php

namespace App\Ai\Agents;

use Laravel\Ai\Contracts\Agent;
use Laravel\Ai\Promptable;

class DocumentAnalyzer implements Agent
{
    use Promptable;

    /**
     * Get the instructions that the agent should follow.
     */
    public function instructions(): string
    {
        return 'You are a document analysis assistant. When given text or a
            . 'provide a concise summary, identify the key topics discussed,
            . 'rate the overall sentiment from 1 (very negative) to 10 (very
            . 'and list any action items or next steps mentioned in the cont
            . 'Be specific and practical in your analysis.';
    }
}
```

Copy

That's it. The agent class encapsulates everything: your system prompt, tool configuration, output schema. When I built StudyLab, I had to wire all this up manually. Custom service classes, response parsing, error handling. The agent pattern wraps it into one clean, reusable class.

Taylor actually mentioned in the livestream that agents were his favorite part of the SDK. He designed them to feel like Laravel Actions or Jobs, a self-contained unit you use

Let's test it. Quickest way is a route:

```php
use App\Ai\Agents\DocumentAnalyzer;

Route::post('/analyze', function (Request $request) {
    $response = (new DocumentAnalyzer)->prompt($request->input('text'));

    return ['analysis' => (string) $response];
});
```

Hit that endpoint with some text and you'll get back an AI-generated analysis. Works. But the response is just a big string. Not super useful if you want to store specific fields in a database or display them in a structured UI.

That's where structured output comes in.

Step 2: Add Structured Output

This is the feature that would have saved me the most time on past projects. With ReplyGenius, I needed the AI to return response options in a specific format: tone, length, the actual reply text. I had to write custom JSON parsing, handle when the model returned malformed JSON (which happens more than you'd think), and validate every field manually.

The SDK handles all of that. Your agent implements the `HasStructuredOutput` interface and defines a `schema` method right inside the class. No separate schema files needed:

```php
<?php

namespace App\Ai\Agents;

use Illuminate\Contracts\JsonSchema\JsonSchema;
use Laravel\Ai\Contracts\Agent;
use Laravel\Ai\Contracts\HasStructuredOutput;
use Laravel\Ai\Promptable;

class DocumentAnalyzer implements Agent, HasStructuredOutput
{
```

```
    public function instructions(): string
    {
        return 'You are a document analysis assistant. Analyze the given con
            . 'and return a structured analysis with summary, topics, sentim
            . 'and action items. Be specific and practical.';
    }

    public function schema(JsonSchema $schema): array
    {
        return [
            'summary' => $schema->string()->required(),
            'topics' => $schema->array()->required(),
            'sentiment' => $schema->integer()->min(1)->max(10)->required(),
            'action_items' => $schema->array()->required(),
        ];
    }
}
```

Now when you prompt the agent, you get back a `StructuredAgentResponse` that you can access like an array:

```
$result = (new DocumentAnalyzer)->prompt($text);

$result['summary'];      // "The document discusses quarterly sales performa
$result['topics'];       // ["Q4 revenue", "customer churn", "expansion plan
$result['sentiment'];    // 7
$result['action_items']; // ["Schedule follow-up with enterprise team", "Rev
```

No JSON parsing. No validation. No "the model sometimes wraps the response in markdown code blocks" headaches. You define your schema with the fluent `JsonSchema` builder, and the SDK guarantees you get back data that matches it.

During development, I usually inspect the raw response with a JSON formatter to make sure the schema is producing what I expect. Old habit from debugging OpenAI responses, but still useful.

Step 3: Accept File Attachments

The SDK makes this simple with the `attachments` parameter:

```php
use App\Ai\Agents\DocumentAnalyzer;

Route::post('/analyze', function (Request $request) {
    $request->validate([
        'document' => 'required|file|max:10240', // 10MB max
    ]);

    $result = (new DocumentAnalyzer)->prompt(
        'Analyze this document thoroughly.',
        attachments: [
            $request->file('document'),
        ],
    );

    return [
        'summary' => $result['summary'],
        'topics' => $result['topics'],
        'sentiment' => $result['sentiment'],
        'action_items' => $result['action_items'],
    ];
});
```

Copy

You can pass uploaded files directly. The SDK handles sending them to the provider in the right format. If you're dealing with larger files, check out my guide on handling large file uploads in Laravel for tips on chunking and validation.

You can also attach multiple files:

```php
$result = (new DocumentAnalyzer)->prompt(
    'Compare these two documents and highlight the differences.',
    attachments: [
        $request->file('document_v1'),
        $request->file('document_v2'),
    ],
);
```

Copy

Step 4: Stream the Response

Here's the thing about AI responses. Users hate staring at a loading spinner for 5-10 seconds. They expect the ChatGPT-style experience where text appears progressively. Taylor specifically mentioned this in the livestream. Streaming was a priority because that's what users expect now.

Swap `->prompt()` for `->stream()`:

```
use App\Ai\Agents\DocumentAnalyzer;

Route::post('/analyze', function (Request $request) {
    return (new DocumentAnalyzer)->stream(
        'Analyze this document.',
        attachments: [$request->file('document')],
    )->then(function ($response) {
        // This runs after the stream completes
        // Save to database, send notification, whatever
        logger('Analysis complete', [
            'text' => $response->text,
        ]);
    });
});
```

The `->stream()` method returns an SSE (server-sent events) response. On the frontend, you consume it depending on your stack:

- → **Livewire**: Use the `wire:stream` directive
- → **Vue/React with Inertia**: Use the `useStream` hook
- → **Vanilla JS**: Standard `EventSource` API

The `->then()` callback is super handy. It runs after all chunks have been streamed. Perfect for saving the final result to your database or firing off a notification. You get the complete response object there with the full text, usage stats, and all streamed events.

I'll be honest, setting up SSE on the frontend used to be annoying. Handling reconnection, parsing chunks, dealing with browser compatibility. The SDK abstracts the backend part

Step 5: Write Tests

This is the part most tutorials skip. Don't skip it.

Testing AI features used to be genuinely hard. The responses aren't deterministic. You can't assert that the model will return the exact same text twice. Taylor actually made this a priority for the SDK because he felt testing was something the community kept overlooking with AI integrations.

The solution is clean:

```php
<?php

use App\Ai\Agents\DocumentAnalyzer;

test('it analyzes a document and returns structured output', function () {
    // Auto-generates fake data matching your schema definition
    DocumentAnalyzer::fake();

    $response = $this->postJson('/analyze', [
        'document' => UploadedFile::fake()->create('report.pdf', 100),
    ]);

    $response->assertOk();
    $response->assertJsonStructure([
        'summary',
        'topics',
        'sentiment',
        'action_items',
    ]);
});
```

`DocumentAnalyzer::fake()` intercepts the agent call and auto-generates fake data that matches your schema definition. No API calls. No flaky tests. No burning through your OpenAI credits during CI runs.

You can also pass specific responses if you need to test exact values:

```
    'The document covers Q4 sales performance.',
]);
```

Or dynamically handle prompts with a closure:

Copy

```
use Laravel\Ai\Prompts\AgentPrompt;

DocumentAnalyzer::fake(function (AgentPrompt $prompt) {
    return 'Response for: ' . $prompt->prompt;
});
```

This is so much better than what I used to do. On StudyLab, my test suite had a `MockOpenAiClient` class with 200+ lines of response fixtures. All of that is now `DocumentAnalyzer::fake()`. One line.

Bonus: Queue It for Background Processing

Not every analysis needs to happen in real time. Maybe you're processing a batch of documents overnight. Or the uploaded file is huge and you don't want the user waiting.

The SDK integrates directly with Laravel's queue system:

Copy

```
(new DocumentAnalyzer)->queue(
    'Analyze this quarterly report in detail.',
    attachments: [$storedFilePath],
)->then(function ($response) use ($document) {
    $document->update([
        'summary' => $response['summary'],
        'topics' => $response['topics'],
        'analyzed_at' => now(),
    ]);

    $document->user->notify(new AnalysisComplete($document));
})->catch(function (Throwable $e) use ($document) {
    logger()->error("Analysis failed for document {$document->id}", [
        'error' => $e->getMessage(),
```

The `->queue()` method dispatches a background job. The user gets an immediate response ("We're processing your document"), and the analysis happens in the background. When it's done, the `->then()` callback fires. If something goes wrong, `->catch()` handles it.

If you're running heavy AI workloads through queues, my post on processing 10,000 queue jobs without breaking covers the infrastructure side of things.

One more thing worth mentioning: provider failover. If you're running AI features in production and uptime matters, you can pass multiple providers:

```
$result = (new DocumentAnalyzer)->prompt(
    'Analyze this document.',
    provider: ['openai', 'anthropic'],
);
```

Copy

If OpenAI is down or rate-limited, it automatically falls back to Anthropic. No custom retry logic needed.

What's Next?

We covered the core workflow: agents, structured output, attachments, streaming, testing, and queuing. But the SDK has a lot more. Conversation memory with `RemembersConversations`, vector stores for semantic search, built-in tools like `FileSearch` and `WebSearch`, image generation, text-to-speech, transcription.

In the livestream, Josh actually built a voice assistant from scratch using Claude Code. It transcribed audio input, queried local markdown files as context, and responded with ElevenLabs text-to-speech. All using the SDK. Took about 30 minutes with the AI doing most of the coding.

The SDK is still early (v0.x), so expect features to ship fast. Agent orchestration (agents calling other agents) isn't supported yet but is on the roadmap. Keep an eye on the official docs and the GitHub repo for updates.

Can I use this with Anthropic or Gemini instead of OpenAI?

Yes. The whole point of the SDK is provider abstraction. Change the provider in your `.env` or pass it directly when prompting. Your agent code stays the same.

Is the Laravel AI SDK free?

The SDK itself is free and MIT licensed. You pay for the AI provider's API costs (OpenAI, Anthropic, etc.). If you want to test for free, use Ollama with a local model.

How does this compare to Prism PHP?

The SDK actually builds on top of Prism under the hood. Taylor described the relationship as similar to "query builder and Eloquent." Prism gives you the lower-level API calls. The SDK adds agents, structured output, conversation memory, testing helpers, and tighter Laravel integration on top.

Can agents call other agents?

Not yet. Agent orchestration (a parent agent delegating to sub-agents) isn't available in the current version. It's been discussed and will likely come in a future release.

Does it work with Ollama for local development?

Yes. Ollama is a supported provider. Install it, pull a model, and point your config at it. Great for development without burning API credits.

Wrapping Up

In about 30 minutes, you went from `composer require` to a working document analyzer with structured output, file attachments, streaming, and test coverage. Plus background processing and provider failover as a bonus.

The Laravel AI SDK makes AI a first-class feature of your application, not something bolted on with HTTP calls and string parsing. If you're building anything that touches AI in Laravel, this is the starting point now.

Got a Product Idea?

I build MVPs, web apps, and SaaS platforms in 7 days. Fixed price, real code, deployed and ready to use.

Book a Free Call | **View My Work**

Currently available for 2-3 new projects

About Hafiz

Full Stack Developer from Italy. I help founders turn ideas into working products fast. 9+ years of experience building web apps, mobile apps, and SaaS platforms.

View My Work →

Get web development tips via email

Join **50+** developers • No spam • Unsubscribe anytime

your@email.com **Subscribe**

Related Articles

Laravel AI SDK: What It Changes, Why It Matters, and Should You Use It?

Laravel's first-party AI SDK is here. Here's what actually changed, how it compa...

Stop Vibe Coding Your Production Apps: A Case for Developer-Driven AI

Vibe coding is great for prototypes. But your production app deserves better. He...

Laravel Search in 2026: Full-Text, Semantic, and Vector Search Explained

Laravel 12 now has built-in support for full-text, semantic, and vector search....

 | About

English | Italiano